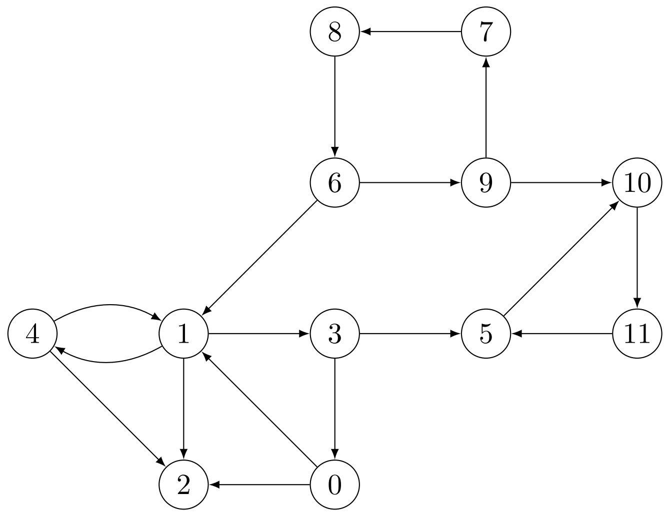
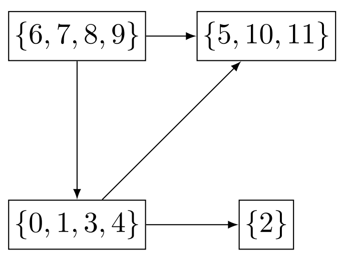
Corrigé TP10

Dans tout le corrigé, n désigne le nombre de sommets d'un graphe G quelconque et p son nombre d'arêtes.

1. Le graphe G_{ex} est le suivant :



   Un parcours en profondeur permet de classer les sommets par ordre de date de fin de traitement décroissante comme suit : $6, 9, 7, 8, 0, 1, 4, 3, 5, 10, 11, 2$. En utilisant cet ordre pour effectuer un parcours en profondeur sur le transposé de G_{ex}, on obtient le graphe quotient suivant :



2. C'est immédiat : on parcourt toutes les arêtes du graphe initial et pour chacune, on ajoute son opposé au graphe en cours de construction. La création de `gt` est en $O(n)$ et la i-ème itération de la boucle principale coûte un $O(\deg(i))$ ce qui implique que le coût de cette boucle est en $O(p)$ puis que la complexité temporelle de `transposer` est en $O(n + p)$.

```
let transposer (g:graphe) :graphe =
  let n = Array.length g in
  let gt = Array.make n [] in
  for i = 0 to n-1 do
    List.iter (fun sommet -> gt.(sommet) <- i::gt.(sommet)) g.(i)
  done;
  gt
```

3. Comme tout parcours, il faut être capable d'implémenter ce genre de fonction en 5 minutes. L'initialisation de `vus` nécessite $O(n)$ opérations puis chaque arête est traitée exactement une fois (grâce au tableau `vus` qui permet d'éviter d'en reconsidérer une) et ce en temps constant. On en déduit une complexité temporelle en $O(n + p)$ à nouveau.

```
let ordre_fin_traitement (g:graphe) :int list =
  let n = Array.length g in
  let vus = Array.make n false and fins = ref [] in
  let rec parcours_profondeur (s:int) :unit =
    if not vus.(s) then
      begin
        vus.(s) <- true;
        List.iter parcours_profondeur g.(s);
        fins := s::(!fins);
      end;
  in
  for i = 0 to n-1 do
    parcours_profondeur i
  done;
  !fins
```

4. Il n'y a pas de grande différence avec la fonction précédente. Le tableau accessibles_depuis, qui contiendra le résultat final, permet également de tester si un sommet a déjà été découvert lors du parcours. La complexité de cette fonction est encore en $O(n+p)$.

```
let numeroter_accessibles (g:graphe) (o:int list) :int array =
  let n = Array.length g in
  let accessibles_depuis = Array.make n (-1) in
  let rec parcours_prof (racine:int) (s:int) :unit =
    (*numérote tous les accessibles depuis le sommet s par racine*)
    if accessibles_depuis.(s) = (-1) then
      begin
        accessibles_depuis.(s) <- racine;
        List.iter (parcours_prof racine) g.(s);
      end
  in
  let numeroter_depuis_s (s:int) :unit = parcours_prof s s in
  (*numérote les sommets accessibles depuis s et non vus par s*)
  begin
    List.iter numeroter_depuis_s o;
    accessibles_depuis
  end
```

5. On applique l'algorithme de Kosaraju. D'après la complexité des fonctions transposer, ordre_fin_traitement et numeroter_accessibles, cette fonction est en $O(n+p)$.

```
let kosaraju (g:graphe) :int array =
  let gt = transposer g and o = ordre_fin_traitement g in
  numeroter_accessibles gt o
```

kosaraju g_ex renvoie le tableau $[0, 0, 2, 0, 0, 5, 6, 6, 6, 6, 5, 5]$ ce qui est cohérent avec la question 1.

Remarque 1 : Comme vu en cours, l'algorithme de Kosaraju fait mieux que calculer les composantes fortement connexes, il permet en fait de les calculer dans un ordre topologique. Pour obtenir cette information, il suffirait de modifier numeroter_accessibles *en attribuant à la première composante découverte* *le numéro* 0*, à la deuxième le numéro* 1*... Le numéro de la composante indiquerait alors sa position dans un ordre topologique dans le graphe des composantes fortement connexes.*

Remarque 2 : Le tableau calculé par **kosaraju** *permet de savoir efficacement si deux sommets sont dans la même composante fortement connexe du graphe considéré ou non.*

6. La fonction max_tableau renvoie la valeur maximale d'un tableau dont les éléments sont des entiers naturels. La fonction nb_cases_non_nulles renvoie le nombre de cases qui ne contiennent pas 0 dans un tableau d'entiers naturels.

   Après avoir calculé pour chaque sommet i le numéro de sa composante fortement connexe grâce à kosaraju, on détermine pour chaque sommet i le nombre de sommets qui ont i comme numéro de composante fortement connexe en parcourant cfc ce qui se fait en $O(n)$. Le tableau obtenu, nommé tailles_cfc n'a plus qu'à être parcouru une fois par max_tableau et une fois nb_cases_non_nulles pour obtenir les deux résultats souhaités, ce en $O(n)$ à nouveau.

   La complexité totale de caracteristiques_graphe_quotient est donc majorée par celle de kosaraju, en $O(n+p)$.

```
let max_tableau (t:int array) = Array.fold_left max 0 t;;
let nb_cases_non_nulles (t:int array) =
  Array.fold_left (fun acc x -> if x > 0 then acc+1 else acc) 0 t;;

let caracteristiques_graphe_quotient (g:graphe) :int*int =
  let n = Array.length g in
  let cfc = kosaraju g in
  let tailles_cfc = Array.make n 0 in
  for i = 0 to n-1 do
    tailles_cfc.(cfc.(i)) <- tailles_cfc.(cfc.(i)) +1
  done;
  (nb_cases_non_nulles tailles_cfc, max_tableau tailles_cfc)
```

   Remarque : On pourrait ne faire qu'un seul parcours du tableau **tailles_cfc** *en calculant lors du même passage le maximum de ce tableau et son nombre de cases non nulles mais ça ne change rien à la complexité temporelle asymptotiquement.*

7. La correction de l'algorithme de Kosaraju repose sur l'étude des dates de fins de traitement lors du premier parcours en profondeur donc il est impératif que ordre_fin_traitement utilise un parcours en profondeur. Le parcours utilisé dans numeroter_accessibles n'a en revanche pas d'importance.

8. Immédiat.

9. On commence par récupérer le nombre n de sommets et p d'arêtes du graphe en utilisant en guise de motif %d %d\n par définition du format utilisé. Attention à ne pas mettre d'espace supplémentaire dans cette chaîne, en particulier juste avant le caractère \n, sans quoi, le motif recherché par Scanf.scanf et la forme de la première ligne ne correspondront pas. La connaissance de n permet d'initialiser les listes d'adjacence du graphe à construire et il ne reste plus qu'à lire les p lignes suivantes pour en déterminer les arêtes.

```
let lire_graphe () :graphe =
  let n,p = Scanf.scanf "%d %d\n" (fun x y -> (x,y)) in
  let g = Array.make n [] in
  for i = 0 to p-1 do
    Scanf.scanf "%d %d\n" (fun x y -> g.(x) <- y::g.(x))
  done;
  g
```

Evidemment, on vérifie que lire_graphe s'exécute correctement sur exemple.txt.

10. On ajoute au code source Ocaml les lignes suivantes :

```
let g = lire_graphe();;
let n,t = caracteristiques_graphe_quotient g in
  Printf.printf "nb_CFC = %d, taille_CFC_max = %d \n" n t
```

Puis on compile pour obtenir un exécutable kosaraju. Il ne reste plus dans un terminal qu'à rediriger le contenu du fichier arxiv.txt via

cat arxiv.txt | ./kosaraju

et on constate que le graphe représenté par arxiv.txt a 21608 composantes fortement connexes dont la plus grande contient 12711 sommets. Et c'est là qu'on est content d'avoir un algorithme linéaire !